Module 2: Making Decisions

Exercise

If you have not already, get prepared for class by downloading the start code:

```
!wget https://student.cs.uwaterloo.ca/~cs114/src/module-02-start.ipynb
```

Discuss the previous module with your neighbour.

- What are all the parts we need to define a function properly?

What does “<” mean?

Consider the expression “$x < 5$”.

In math class, it tells us something about x.

We might combine the statement “$x < 5$” with the statements “x is even”, “x is positive” and “x is a perfect square” to conclude “x is 4”.

In Python, “<” means something different. A variable such as `x` already has a value.

What does “<” mean?

Suppose I define a constant: `x = 10`

Now I create a Python expression as close as possible to the math expression “$x > 5$”:

```
x > 5
```

This is **asking** “is this true?”

The statement “$x > 5$” can only be true or false. Which one?

Since `x` refers to the value `10`, saying `x > 5` is like saying `10 > 5`.

Since it is true that $10 > 5$, the statement evaluates to **`True`**.

On the other hand, if I create a variable:

```
y = 2
```

Now `y > 5` ⇒ `2 > 5` ⇒ **`False`** since it is not true that $2 > 5$.

Boolean values (`bool`)

`True` and `False` are each values, just like `42`, `3.14`, and `"hello world"` are values.

These values are called **Booleans**, after George Boole. The Python type is `bool`.

`<`, `>`, `<=`, `>=`, `==`, and `!=`, are operators, each of which results in a Boolean value.

- `5 < 10` ⇒ `True`
- `20 < 10` ⇒ `False`
- `42 == 42` ⇒ `True`

The `!=` operator is supposed to look a little like "≠". It means "not equal".

- `42 != 17` ⇒ `True`
- `42 != 42` ⇒ `False`

A `bool` value is a value; it can be stored in a variable:

```
x = y < 4
```

Provided `y` is a number less than 4, `x` will now be `True`; if `y` is a number 4 or greater, `x` will now be `False`.

Exercise

What is the value of `x` after I run these two lines of code?

```
x = 5
x = x == 5
```

`and` now for something completely different

We combine Boolean expressions using the operators `and`, `or`, and `not`. These all take and return `bool` values.

- `and` returns `False` if at least one of its arguments is `False`, and `True` otherwise.
  - `(5 > 4) and (7 != 2)` ⇒ `True`
  - `(5 >= 5) and (7 <= 2) and (5 > 1)` ⇒ `False`
  - `True and (3 < 7) and (9 >= 1)` ⇒ `True`
- `or` returns `True` if at least one of its arguments is `True`, and `False` otherwise.
  - `(5 >= 4) or (7 > 2)` ⇒ `True`
  - `(4 > 5) or (2 != 2) or (9 < 4)` ⇒ `False`
- `not` returns `True` if its argument is `False`, and `False` if its argument is `True`.
  - `not (5 == 4)` ⇒ `True`
  - `not ((10 <= 15) and (7 > 3))` ⇒ `False`

Work out what the snippet should display. Then run to verify.

Exercise

```
def foo(a: int, b: int) -> bool:
    return a == 3 or b == 3
foo(3, 3) ⇒ ?
foo(6, 7) ⇒ ?
foo(3, 7) ⇒ ?
```

Work out what the snippet should display. Then run to verify.

Exercise

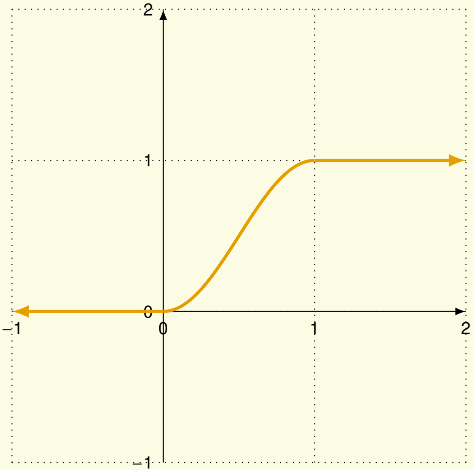
```
def bar(a: int, b: int) -> bool:
    return a or b == 3
bar(0, 3) ⇒ ?
bar(3, 0) ⇒ ?
bar(5, 5) ⇒ ?
```

!

We write `a or b == 3`.
In English I might say "if *a* or *b* is three". That is not what the Python code means.
This is like saying `(a) or (b == 3)`. The value is `True` if `a` is `True`, or if `b == 3` is `True`.
It does **not** mean the value is `True` if `a == 3` is `True` or if `b == 3` is `True`.

Python is not English! For clarity, use `and` and `or` with values that are `True` or `False`.

To get the meaning of the English, we should write `a == 3 or b == 3`.



A sin-squared window, used in signal processing, can be described by the following piecewise function:

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 1 \\ \sin^2(x\pi/2) & \text{for } 0 \leq x < 1 \end{cases}$$

Under some conditions, it does one thing; under other conditions, it does other things.

Using `if/elif/else` to describe cases

We can write this mathematical expression in Python as follows:

$$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 1 \\ \sin^2(x\pi/2) & \text{for } 0 \leq x < 1 \end{cases}$$

```python
def ssqw(x: float) -> float:
    """Transform x by a sin-squared window."""
    if x < 0.0:
        return 0.0
    elif x >= 1.0:
        return 1.0
    else:
        return math.sin(x * math.pi / 2) ** 2
```

When working with `if` we write:

1. The keyword `if`,
2. a Boolean expression ("question"),
3. a colon,
4. an indented block of code.

This may be followed by:

5. additional `elif` - Boolean - colon - block,
6. an `else` - colon - block.

Interpreting `if` statements

Python checks the "question" of the `if`. If the question `True`, it executes the block of code.

Otherwise, it looks through any `elif` statements, until it find a branch where the "question" evaluates to `True`.

If none evaluates to `True`, it executes the `else` branch, if there is one.

```
def ssqw(x: float) -> float:
    """Transform x by a sin-squared window."""
    if x < 0.0:
        return 0.0
    elif x >= 1.0:
        return 1.0
    else:
        return math.sin(x * math.pi / 2) ** 2
```

Imagine calculating:

```
ssqw(-1.5)
ssqw(1.5)
ssqw(0.1)
```

Exercise

Use `if` to write a function `absolute_value(n: float) -> float` which returns $|n|$.
(There is a built-in function `abs` which does this, but don't use it now.)

Consider that one way to define absolute value is as follows:

$$|n| = \begin{cases} -n & \text{if } n < 0 \\ n & \text{if } n \geq 0 \end{cases}$$

Nested Conditionals

A museum offers free admission for people who arrive after 5 pm. Otherwise, the cost of admission is based on a person's age: age 12 and under are charged \$9 and everyone else is charged \$16.

We will write a function `admission(isafter5: bool, age: int) -> int` to calculate the admission price.

Exercise

Use `check.expect` to write at least 3 tests for `admission`, one for each price category.

Exercise

Complete the function `admission(isafter5: bool, age: int) -> int` that returns the admission cost.

Hint

`isafter5` is a `bool`.
So it can be directly used as a question in an `if` statement, like `if isafter5:`

Flattening Nested Conditionals

Sometimes it is desirable to flatten conditionals.

That is, instead of having a **if** with another **if** inside, we can rework them so they are multiple clauses of a single **if**.

```python
def cost(isafter5: bool, age: int) -> int:
    if isafter5:
        return 0
    else:
        if age <= 12:
            return 9
        else:
            return 16
```

$\leftrightarrow$

```python
def cost(isafter5: bool, age: int) -> int:
    if isafter5:
        return 0
    elif age <= 12:
        return 9
    else:
        return 16
```

Flattening Nested Conditionals

Exercise

Flatten the code in this function so there is only one `if/elif/else`, with four branches.

```
def flatten_me(x: int) -> str:
```

> *"In science, computing, and engineering, a **black box** is a device...which can be viewed in terms of its inputs and outputs, **without any knowledge of its internal workings**." (Wikipedia)*

Black-box testing refers to testing **without reference to how the program works.** Black-box tests should be written before you write your code. Your examples are black-box tests.

> *"A white box is a subsystem whose internals can be viewed but usually not altered." (Wikipedia)*

White-box testing should exercise every line of code. Design a test to check both sides of every `question` in every **`if`/`elif`/`else`**.

These tests are designed after you write your code, by **looking at how the code works**.

White-Box Testing

Consider writing white box tests for this code:

```
def cost(isafter5: bool, age: int) -> int:
    if isafter5:
        return 0
    elif age <= 12:
        return 9
    else:
        return 16
```

We need to be sure to test every branch. Here is one suggestion of tests:

1. `check.expect(cost(True, 42), 0)` to test the first branch.
2. `check.expect(cost(False, 7), 9)` to test the second branch.
3. `check.expect(cost(False, 42), 15)` to test the third branch.

Additional tests are desirable to check **edge cases**. These help us verify that we did what we meant to do: did we really mean to use `<=` instead of `<` ?

Testing with `age` of `11`, `12`, and `13` would cover the edge cases.

Boolean operators and strings

We can use the same operators on any type, not just on numbers. Try these:

```
str1 = "Frobisher"
str2 = "Frontenac"
```

```
print(str1 == str2)
```

Two strings are equal if they're the same.

For strings, “<” compares character by character. If the strings start the same, it goes until it finds something different.

So because `'b' < 'o'` ⇒ `True`, we see also that `'Frobisher' < 'Frontenac'` ⇒ `True`.

One detail: every uppercase letter is “less than” every lowercase: `'Z' < 'a'` ⇒ `True`.

We'll call the ordering that we get from < “alphabetic order”, even when we're not comparing alphabetic letters: `"$1ll1n355" < "&t1t00d"` ⇒ `True`

Module summary

- Become comfortable using Boolean operators such as `<`, `>=`, `==`, and `!=`.
- Start using **`if`/`elif`/`else`**, **`and`**, **`or`**, and **`not`**.
- Get used to combining these statements with the rest of our tools.
- Test these expressions, and know what black-box and white-box testing are.

Before we begin the next module:

- Read and complete the exercises in module 2 of the online textbook, at https://online.cs.uwaterloo.ca/
- Complete the module 2 Review Quiz, due on Monday.